EAST Search History

Ref #	Hits	Search Query	DBs	Default Operator	Plurals	Time Stamp
S2	1	2004\0013307	US-PGPUB; USPAT	OR	ON	2007/07/26 19:46
S5	1	2004\0054692	US-PGPUB; USPAT	OR	ON	2007/07/30 15:23
S6	1	"7043686".PN.	US-PGPUB; USPAT	OR	ON	2007/07/30 15:23
S8	4747	(decod$3 encod$3) with (XML "eXtensible Markup Language")	US-PGPUB; USPAT	OR	ON	2007/07/30 16:48
S9	579	(decod$3 encod$3) with (XML "eXtensible Markup Language") with (schema XSD DTD)	US-PGPUB; USPAT	OR	ON	2007/07/30 16:48
S10	7	(decod$3 encod$3) with (XML "eXtensible Markup Language") with (schema XSD DTD) with normaliz$5	US-PGPUB; USPAT	OR	ON	2007/07/30 16:49
S11	5	(decod$3 encod$3) with (XML "eXtensible Markup Language") with (schema XSD DTD) with (meta$1schema (schema ADJ for ADJ schemas))	US-PGPUB; USPAT	OR	ON	2007/07/30 16:51
S12	14	((decod$3 encod$3) with ((schema XSD DTD) (XML "eXtensible Markup Language"))) AND (meta$1schema (schema ADJ for ADJ schemas))	US-PGPUB; USPAT	OR	ON	2007/07/30 16:52
S13	8	(decod$3 and encod$3 and ((schema XSD DTD) and (XML "eXtensible Markup Language"))) AND (meta$1schema (schema ADJ for ADJ schemas))	US-PGPUB; USPAT	OR	ON	2007/07/31 15:38
S14	1	(decod$3 and encod$3 and ((schema XSD DTD) and (XML "eXtensible Markup Language"))) and ((anon anonim$3) with type)	US-PGPUB; USPAT	OR	ON	2007/07/31 15:42
S15	0	("7043686").URPN.	USPAT	OR	ON	2007/07/31 15:43
S16	19	("20020120616" | "20040013307" | "5438512" | "5557720" | "5583762" | "5630126" | "5819264" | "5915259" | "5933842" | "5999949" | "6011905" | "6016467" | "6182029" | "6330574" | "6336214" | "6523172" | "6635088" | "6647534" | "6665665").PN.	US-PGPUB; USPAT; USOCR	OR	ON	2007/07/31 15:43

EAST Search History

S17	20	(anon$ with type) with (XML (eXtensible ADJ Markup ADJ Language))	US-PGPUB; USPAT; USOCR	OR	ON	2007/07/31 17:05
S18	6	((transform$5 normaliz$5) with (anon$ with type)) and (XML (eXtensible ADJ Markup ADJ Language))	US-PGPUB; USPAT; USOCR	OR	ON	2007/07/31 17:10
S19	5	schema near5 branch near2 code near10 length	US-PGPUB; USPAT; USOCR	OR	ON	2007/08/02 18:05
S20	1	(byte near2 code) near5 display near5 (path paylod)	US-PGPUB; USPAT; USOCR	OR	ON	2007/08/02 18:41
S21	1	(byte near2 code) SAME display near5 (path paylod)	US-PGPUB; USPAT; USOCR	OR	ON	2007/08/02 18:42
S22	1	(byte near2 code) SAME display near10 (path paylod)	US-PGPUB; USPAT; USOCR	OR	ON	2007/08/02 18:42
S23	15	(byte near2 code) SAME (display SAME (path paylod))	US-PGPUB; USPAT; USOCR	OR	ON	2007/08/02 18:42
S24	2291	(707/101).CCLS.	USPAT; USOCR	OR	OFF	2007/08/02 19:45
S25	586	(decod$3 encod$3) with (XML "eXtensible Markup Language") with (schema XSD DTD)	US-PGPUB; USPAT	OR	ON	2007/08/02 19:45
S26	12	S24 and S25	US-PGPUB; USPAT	OR	ON	2007/08/02 19:45

Canonical XML forms

for post-schema-validation infosets

A preliminary reconnaissance

C. M. Sperberg-McQueen

24 April 2002

This document describes one possible way to provide access to the post-schema-validation information set: by defining canonical XML forms both (1) for the original document together with the 'infoset enhancements' (type attribution, validity information, etc.) and (2) for the schema components themselves.

This document is written with the intention of submitting it to the XML Schema WG as a working paper; it has not been approved by that WG or anyone else and has no official status of any kind. It has benefitted from the author's conversations with Dave Hollander, Philippe Le Hégaret, and Peter Fankhauser, to whom thanks.

1. Introduction

There may be many things we expect from schema processors in practice, but the W3C XML Schema 1.0 specification [Thompson et al. 2001] specifies only a single process, namely *schema-validity assessment*, which is defined as a mapping from an input information set (infoset) to a richer output information set.

The input information set is an instance of the standard infoset defined by the XML Information Set specification [Cowan/Tobin 2001]. We will refer to the infoset defined by the XML Information Set specification as the *core infoset*.

The output information set contains the input information set as a proper subset, and adds further information about the datatypes and validity of elements and attributes, and so on.

The information in the post-schema-validation information set may usefully be partitioned into three sets of information items and properties:

- the input information set, which we assume we wish to make available without loss of information
- 'enhancements' to the input information set; these take the form of
  - new properties on existing information items
  - new information items of core infoset types (for example, if the schema declares a default value for an element not present in the input infoset, a new attribute information item will be constructed)
  - new information items of new types not defined in the XML Information Set specification [Cowan/Tobin 2001]; see below for further discussion.
- schema components, represented as new kinds of information items; N.B. processors are not required to provide these; they may instead provide the properties [type definition name], [type definition namespace], and [type definition anonymous]. We expect that lightweight processors may prefer to provide just the type names instead of the types themselves.

This document considers design questions associated with making this information accessible in canonical XML forms.[1] It first discusses possible methods of providing the infoset enhancements, then methods of providing access to schema components, and in conclusion considers some issues related to streaming processors.

2. Infoset enhancements

2.1. A simple approach

One simple approach to providing an XML form for infoset enhancements is shown by the use of *xsi:type* as already defined in the XML Schema specification.

If we used the *xsi:type* attribute to record the type associated by the schema with each element[2] in the document, part of the sample purchase order in the XML Schema tutorial might look like this:

```
<shipTo country="US"
        xsi:type="po:USAddress">
 <name   xsi:type="xsd:string">Alice Smith</name>
 <street xsi:type="xsd:string">123 Maple Street</street>
 <city   xsi:type="xsd:string">Mill Valley</city>
 <state  xsi:type="xsd:string">CA</state>
 <zip    xsi:type="xsd:decimal">90952</zip>
</shipTo>
```

The basic approach outlined here is just an extension and modification of this basic idea.

First, in order to avoid any confusion between any *xsi:type* attributes in the input and the attributes supplied as part of an XML serialization of the PSVI, we might not want to use the *xsi:type* attribute itself.[3] In the examples in this document, all the attributes added as part of re-serializing the PSVI will

be in the *http://www.example.com/psvi* namespace, with the prefix *psvi*. Our example now might look like this:

```
<shipTo country="US"
        psvi:type="po:USAddress">
 <name   psvi:type="xsd:string">Alice Smith</name>
 <street psvi:type="xsd:string">123 Maple Street</street>
 <city   psvi:type="xsd:string">Mill Valley</city>
 <state  psvi:type="xsd:string">CA</state>
 <zip    psvi:type="xsd:decimal">90952</zip>
</shipTo>
```

Next, note that the XML Schema specification identifies a number of information set contributions to be attached to element information items, not just the [type definition name] property conveyed by *xsi:type* and *psvi:type*. For the sake of simplicity in exposition, for now let us consider just a few such properties, namely the properties

- [member type definition] (for an element valid against a union type, the member type definition against which it was successfully validated),
- [validation attempted] ("full", "none", or "partial")[4]
- [validation context] (the element information item at which validation started), and
- [validity] ("valid", "invalid", or "notKnown")[5]

If we use attributes similar to *psvi:type* to record these properties of elements in the document, and change one of the values to be illegal, for variety's sake, our sample might look like this:

```
<shipTo country="US"
        psvi:type="po:USAddress"
        psvi:validation-attempted="full"
        psvi:validation-context="e1"
        psvi:validity="valid">
 <name psvi:type="xsd:string"
       psvi:validation-attempted="full"
       psvi:validation-context="e1"
       psvi:validity="valid">Alice Smith</name>
 <street psvi:type="xsd:string"
         psvi:validation-attempted="full"
         psvi:validation-context="e1"
         psvi:validity="valid">123 Maple Street</street>
 <city psvi:type="xsd:string"
       psvi:validation-attempted="full"
       psvi:validation-context="e1"
       psvi:validity="valid">Mill Valley</city>
 <state psvi:type="xsd:string"
        psvi:validation-attempted="full"
        psvi:validation-context="e1"
        psvi:validity="valid">CA</state>
 <zip psvi:type="xsd:decimal"
      psvi:validation-attempted="full"
      psvi:validation-context="e1"
      psvi:validity="invalid">90952-9765</zip>
</shipTo>
```

An XML document to which special attributes have been added to record PSVI-related information I will call a *PSVI-decorated document*.

A full working out of this approach to serializing PSVI infoset enhancements will differ from the example just given in two ways. First, there will be additional attributes to cover the other properties of elements in the PSVI. Second, the approach needs to be extended in order to solve a couple of technical problems. The next few sections describe these technical problems and sketch solutions for them.

2.2. Names of types

What happens when a schema defines a complex type locally, within an element declaration? The schema will specify no name for that type: it's illegal to use the *name* attribute on local type declarations.

There are at least three easy solutions to this problem:

- generate a unique name for the anonymous types, as schema processors are allowed to do
- use an XPath expression which points into the canonical XML form of the schema (in this case we'd probably want to do the same for all types including the named ones)
- use the syntax for NUNs (normalized universal names) specified in the current draft of the Formal Description document

Using generated names, we might get something like the following for another part of the sample purchase order:

```
<items psvi:type="po:Items">
 <item partNum="872-AA"
       psvi:type="_anon001">
  <productName psvi:type="xsd:string">Lawnmower</productName>
  <quantity psvi:type="_anon002">1</quantity>
  <USPrice psvi:type="xsd:decimal">148.95</USPrice>
  <comment psvi:type="xsd:string">Confirm this is electric</comment>
 </item>
 <item partNum="926-AA"
       psvi:type="_anon001">
  <productName psvi:type="xsd:string">Baby Monitor</productName>
  <quantity psvi:type="_anon002">1</quantity>
  <USPrice psvi:type="xsd:decimal">39.98</USPrice>
  <shipDate psvi:type="xsd:date">1999-05-21</shipDate>
 </item>
</items>
```

For brevity I've omitted the properties other than *psvi:type*.

If we use XPath expressions pointing in a canonical form of the schema, and if we assume we'll want to use XPath expressions for the named types as well, we might get something like the following:

```
<items psvi:type="//xsd:complexType[@ns='http://www.example.com/PO1'
                   and @name='Items']">
 <item partNum="872-AA"
       psvi:type="//xsd:complexType[@ns='http://www.example.com/PO1'
                   and @name='Items']
                  /xsd:sequence
                  /xsd:element[@name="item"]
                  /xsd:complexType">
  <productName psvi:type="//xsd:simpleType[
                            @ns='http://www.w3.org/2001/XMLSchema'
                            and name='string']">Lawnmower</productName>
  <quantity psvi:type="//xsd:complexType[
```

```
                    @ns='http://www.example.com/PO1'
                    and @name='Items']
                  /xsd:sequence
                  /xsd:element[@name="item"]
                  /xsd:complexType
                  /xsd:element[@name="quantity"]
                  /xsd:simpleType">1</quantity>
  <USPrice psvi:type="//xsd:simpleType[
                          @ns='http://www.w3.org/2001/XMLSchema'
                          and name='decimal']">148.95</USPrice>
  <comment psvi:type="//xsd:simpleType[
                          @ns='http://www.w3.org/2001/XMLSchema'
                          and name='string']"
  >Confirm this is electric</comment>
 </item>
 ...
</items>
```

If we use a syntax for NUNs something like the long syntax of XPath, we might get something like the following:

```
<items psvi:type="/type::po:Items">
 <item partNum="872-AA"
       psvi:type="/type::po:Items/element::item/type::*">
  <productName psvi:type="/type::xsd:string">Lawnmower</productName>
  <quantity psvi:type="/type:po:Items/element::item
                       /element::quantity/type::*">1</quantity>
  <USPrice psvi:type="/type::xsd:decimal">148.95</USPrice>
  <comment psvi:type="/type:xsd:string">Confirm this is electric</comment>
 </item>
 ...
</items>
```

If we use a syntax for NUNs something like that in the current draft of FD, we might get something like the following:

```
<items psvi:type="/%po:Items">
 <item partNum="872-AA"
       psvi:type="/%po:Items/item/%*">
  <productName psvi:type="/%xsd:string">Lawnmower</productName>
  <quantity psvi:type="/type:po:Items/item/quantity/%*">1</quantity>
  <USPrice psvi:type="/%xsd:decimal">148.95</USPrice>
  <comment psvi:type="/%xsd:string">Confirm this is electric</comment>
 </item>
 ...
</items>
```

The choice among these solutions or variations upon them is a design decision the WG will have to make if we go in this direction. Whatever approach is chosen should (I think) work both in cases where the PSVI-decorated document is transmitted and used in isolation and in cases where it is accompanied by an XML representation of the schema components.

2.3. Attaching information to attributes

The second major technical problem with the simple approach outlined above is that *xsi:type* only works for elements. What about types for attributes?

One possible solution is to use the basic idea of the W3C submission on datatypes for DTD notation [Buck et al. 2000], and provide a single attribute, which I'll call *psvi:atttypes* which contains a sequence of (*attribute-name*, *attribute-type-name*) pairs for all of the `normal' attributes on the original element. [6]

Decorated with type information, the beginning of the purchase order might look like this:

```
<purchaseOrder orderDate="1999-10-20"
               psvi:type="po:purchaseOrderType"
               psvi:atttypes="orderDate xsd:date"
               xmlns="http://www.example.com/PO1"
               xmlns:po="http://www.example.com/PO1"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
>
 <shipTo country="US"
         psvi:type="po:USAddress"
         psvi:atttypes="country xsd:NMTOKEN">
  <name psvi:type="xsd:string">Alice Smith</name>
  ...
```

It should be noted that I am fully aware that this solution is ugly. Long meditation on this problem has convinced me that *every* available solution to this problem is ugly: attributes were designed to have atomic or simple list values, not to have attributes of their own, and I no longer expect to find a pretty way to go against the grain of XML here. The notation described is simple to understand, is parallel to the notation used for *xsi:schemaLocation*, has already been defined in a publicly accessible document [Buck et al. 2000], and is at least as good as any alternative I have thought of.

2.4. Full list of properties and information items

As noted above, the PSVI includes a lot of properties beyond those illustrated above. I won't try to work out a full proposal for dealing with all of them here, but it does seem convenient to reproduce the relevant part of the list provided in Appendix C.2 of the Structures spec.

2.4.1. For core information items

The full list of additional properties supplied for core information items by a schema processor (and the information-set contribution rules that define them) is as follows. For each of these items, the definition of a canonical XML form will need to define an attribute to carry the information and rules for conveying the information by means of the attribute value. For example,

For element information items, the additional properties are:

- [element declaration] (Element Declaration)
- [ID/IDREF table] (ID/IDREF Table)
- [identity-constraint table] (Identity-constraint Table)
- [member type definition] (Element Validated by Type)
- [member type definition anonymous] (Element Validated by Type)
- [member type definition name] (Element Validated by Type)
- [member type definition namespace] (Element Validated by Type)
- [nil] (Element Declaration)
- [notation] (Validated with Notation)
- [notation public] (Validated with Notation)

- [notation system] (Validated with Notation)
- [schema default] (Element Validated by Type)
- [schema error code] (Validation Failure (Element))
- [schema information] (Schema Information)
- [schema normalized value] (Element Validated by Type)
- [schema specified] (Element Default Value)
- [type definition] (Element Validated by Type)
- [type definition anonymous] (Element Validated by Type)
- [type definition name] (Element Validated by Type)
- [type definition namespace] (Element Validated by Type)
- [type definition type] (Element Validated by Type)
- [validation attempted] (Assessment Outcome (Element))
- [validation context] (Assessment Outcome (Element))
- [validity] (Assessment Outcome (Element))

For attribute information items:

- [attribute declaration] (Attribute Declaration)
- [member type definition] (Attribute Validated by Type)
- [member type definition anonymous] (Attribute Validated by Type)
- [member type definition name] (Attribute Validated by Type)
- [member type definition namespace] (Attribute Validated by Type)
- [schema default] (Attribute Validated by Type)
- [schema error code] (Validation Failure (Attribute))
- [schema normalized value] (Attribute Validated by Type)
- [schema specified] (Assessment Outcome (Attribute))
- [type definition] (Attribute Validated by Type)
- [type definition anonymous] (Attribute Validated by Type)
- [type definition name] (Attribute Validated by Type)
- [type definition namespace] (Attribute Validated by Type)
- [type definition type] (Attribute Validated by Type)
- [validation attempted] (Assessment Outcome (Attribute))
- [validation context] (Assessment Outcome (Attribute))
- [validity] (Assessment Outcome (Attribute))

2.4.2. Additional information items

Some properties in the infoset enhancement point to information items which are not defined in the core XML infoset. Deciding on the best way of providing XML access to these as part of a PSVI-decorated instance is an open problem. I have no proposals at the moment.

For ID/IDREF binding information items:

- [binding] (ID/IDREF Table)
- [id] (ID/IDREF Table)

For Identity-constraint Binding information items:

- [definition] (Identity-constraint Table) [node table] (Identity-constraint Table)

For namespace schema information information item properties

- [schema components] (Schema Information) [schema documents] (Schema Information) [schema namespace] (Schema Information)

For the schema document information item:

- [document location] (Schema Information)
- [document] (Schema Information)

3. Schema components

Full access to the PSVI by means of XML representations requires an XML representation for schema components. To distinguish this XML representation from the XML representation already defined in the XML Schema Recommendation, I will refer to it as a *schema component dump* format.

I do not propose here to work out a full design for a dump format. It seems clear that several very different designs are possible; my hope here is only to suggest some of the axes of variation and to identify some of the basic design questions.

I assume that a canonical XML form for schema components should have the following characteristics:

- There should be a single XML document for the entire schema, rather than separate documents for each namespace which contributes components to the schema.[7]
- It should be easy (within limits) to construct an XPath which points to any given schema component, without knowledge of the document structure of the transfer-syntax schema documents from which it may have been constructed. It should be easy for us to define a `canonical' XPath expression to be used to identify any construct, thus making schema components `first-class objects' on the Web.[8]
- The canonical form must provide information about the type derivation tree and other semantically important information from the original schema documents, even if such information is no longer essential for validation.

If these are accepted as desiderata or requirements, it is clear that the existing XML transfer syntax is not as it stands a suitable dump format. It does not commingle declarations for components in different namespaces. It has many features for reducing redundancy and improving maintainability, but which make it hard to provide simple rules for constructing XPath expressions to denote arbitrary constructs. The *xsd:attribute* element which declares an attribute, for example, may be a child of the relevant *xsd:complexType* element, or may be very distant from it in the schema document, connected only by a long chain of inheritance and group reference pointers.

I conclude that if we want a dump format with the qualities described, we need to design one; we don't already have one in the spec.

One obvious design question is: should the dump format be made as similar as possible to the existing transfer syntax, or made more similar to the components?

3.1. Dump format based on existing transfer syntax

This approach to the dump format is derived more or less directly from a proposal made by Lee Buck in October of 1999 at our first meeting in Reston. We did not take it up then because it seemed likely to delay the completion of XML Schema 1.0. As I remember it, Lee's proposal was simple: to define a subset of the transfer syntax that eliminated everything that contributed most to making it hard to use

XPointers conveniently and reliably to identify schema constructs.

If we wish to make the dump format resemble the transfer syntax, I think the salient characteristics are that the dump syntax

- should not include inclusions, imports, or redefines
- should not include named groups
- should not include local declarations of elements or types[9]
- must change some of the rules for linking references to their corresponding definitions
- should otherwise be as close to the existing transfer syntax as reasonable

If we stay close to the transfer syntax where possible, while still making all components top-level objects, one of the complex type declarations in the Primer example might turn into something like this:

```
<xsd:complexType id="_ct_Items" name="Items">
 <xsd:sequence>
  <xsd:element ref="_el_item" minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType id="_ct_anon01">
 <xsd:sequence>
  <xsd:element ref="_el_productName"/>
  <xsd:element ref="_el_quantity"/>
  <xsd:element ref="_el_USPrice"/>
  <xsd:element ref="_el_comment"  minOccurs="0"/>
  <xsd:element ref="_el_shipDate" minOccurs="0"/>
 </xsd:sequence>
 <xsd:attribute name="partNum" type="SKU" use="required"/>
</xsd:complexType>

<xsd:simpleType id="_st_anon02">
 <xsd:restriction base="xsd:positiveInteger">
  <xsd:maxExclusive value="100"/>
 </xsd:restriction>
</xsd:simpleType>

<xsd:element id="_el_item" name="item" type="_ct_anon01"/>
<xsd:element name="productName" type="xsd:string"/>
<xsd:element name="quantity" type="_st_anon02"/>
<xsd:element name="USPrice"  type="xsd:decimal"/>
<xsd:element name="shipDate" type="xsd:date"/>
```

I have supplied IDs for anonymous items as needed -- in fact I have supplied IDs for everything, and used ID references to link all references to their components.[10]

3.2. Dump format unlike transfer syntax

It might be preferable to make the dump syntax intentionally unlike the transfer syntax.[11] An XML syntax drawn more or less directly from the existing component + properties description of the component set might produce declarations like these:

```
<df:element-declaration name="item"
                        targetns="http://www.example.com/PO1"
                        typedef="_ct_anon01"
```

```
                        scope="_ct_Items"
                        nillable="false"
                        identity-constraint-defs=""
                        substitution-group-affiliation=""
                        disallowed-substitutions=""
                        abstract="false"
                        >
 <df:annotation> ... </df:annotation>
</df:element-declaration>

<df:complexTypeDefinition id="_ct_anon01"
                          name=""
                          derivation-method="restriction"
                          final=""
                          abstract="false"
                          attributeUses=""
                          attributeWildcard=""
                          prohibitedSubstitutions=""
                          >
 <!--* no name="..." on anonymous types *-->
 <df:element-only/>
 <df:sequence>
  <df:element ref="_el_productName"/>
  <df:element ref="_el_quantity"/>
  <df:element ref="_el_USPrice"/>
  <df:element ref="_el_comment"  minOccurs="0"/>
  <df:element ref="_el_shipDate" minOccurs="0"/>
 </df:sequence>
 <df:attribute name="partNum" type="SKU" use="required"/>
</df:complexTypeDefinition>
```

Some WG members might prefer a dump syntax that more closely resembled the notation used to display component structure in the current Formal Description draft.

3.3. A possible enhancement

Depending on the internal structure eventually chosen for the dump format, XPointer expressions or even NUNs for individual constructs may prove unwieldy. We might well prefer to assign unique local names to every construct, so that they can be referred to using simple QNames. One way to do this would be to include a small table in the dump format which gives the QName of each construct together with an XPath expression that (when applied to the dump-format document itself) will locate the component so named.

4. Special considerations

It is easiest to think about the canonical XML forms for the PSVI as if the infoset enhancements would be in one document, and the schema components in a separate XML document. In some applications, it will also be easiest to work with things this way.

In some applications, however, it will be most convenient if the PSVI-decorated instance and the dump of its schema could be transmitted in a single stream. For this, we may choose to use methods similar to those used to embed a schema in the same document as the material it is describing:

```
<!DOCTYPE envelope:doc-schema-pair [
<!ATTLIST envelope:schema id ID #REQUIRED>
```

```
]>
<envelope:doc-schema-pair xmlns:envelope="...">
 <envelope:schema id="schema">
  <!--* schema in (transfer or) dump format goes here *-->
 </envelope:schema>
 <envelope:instance xsi:schemaLocation="#schema">
  <!--* original instance doc goes here *-->
 </envelope:instance>
</envelope:doc-schema-pair>
```

Note, however, that this requires us to emit the schema in its entirety before we have emitted any of the document instance. It's hard to see how a streaming processor using just-in-time schema component construction could emit such a document.

If we are simply writing to files, this need not be a problem. But if we are sending information from the schema processor to a downstream application in a single datastream, we may need to find a way to interleave instance and schema.

So we may wish to explore the possibility of interleaving parts of the schema components in dump format with the decorated document instance. The decorated document instance comes in this way to resemble the input XML document less and less: we have added a large number of new attributes to the instance already, and interleaving would require that we also add a certain number of elements in the dump-format namespace.

Ideally, we would interleave the two in such a way that any process which correctly handles the input infoset can also handle the single-streamed PSVI; I don't know how to achieve that ideal, but some designs come closer to it than others.

Both the additional attributes and the additional elements may get in the way of normal processing of the instance document (which means they may defeat the purpose of serializing the infoset as a decorated instance rather than using one of the normal forms for dumping graphs into XML). It may be noted, however, that if we add only attributes, then any process which refers to attributes only by name will produce exactly the same results from a document and from its PSVI. Only if the process refers to attributes using a wildcard of some kind will the results differ.

Similarly, if we add schema component elements like *cpsvi:complexType* to the output stream, any process which refers to elements only by name will produce exactly the same results. Any process which refers to wildcards (e.g. a stylesheet which includes an unrestricted *apply-templates*) will encounter unexpected declarations and produce divergent results.

We can thus formulate a simple rule for writing processes which will produce the same results even if a document has been decorated or interleaved with PSVI information: avoid the use of XSLT wildcards (including *xsl:apply-templates* without a *select* attribute) or their equivalents in the implementation language. Alternatively, include special rules to catch attributes and elements in the psvi namespace, and handle them appropriately. (Also need to handle annotation correctly.)

5. Conclusion

This document identifies some design questions we must face if we wish to specify canonical XML forms for the post-schema-validation information set. I think it establishes fairly clearly that such canonical XML forms can be specified in such a way as to make them useful; it also identifies some design questions which can usefully be answered by a design. This document thus can be used to argue that defining canonical XML forms for the PSVI is feasible.

The next step is to explore other possibilities for specifying such canonical forms and decide on an

overall design direction.

A. References

[Boyer 2001] Boyer, John. 2001. *Canonical XML Version 1.0*. W3C Recommendation 15 March 2001. [Cambridge, Sophia-Antipolis, and Tokyo]: World Wide Web Consortium. http://www.w3.org/TR/xml-c14n

[Buck et al. 2000] Buck, Lee, Charles F. Goldfarb, and Paul Prescod, ed. *Datatypes for DTDs (DT4DTD) 1.0*. W3C Note 13 January 2000. [Cambridge, Sophia-Antipolis, and Tokyo]: World Wide Web Consortium. http://www.w3.org/TR/dt4dtd

[Cowan/Tobin 2001] Cowan, John, and Richard Tobin, ed. 2001. "XML Information Set." W3C Recommendation 24 October 2001. [Cambridge, Sophia-Antipolis, Tokyo]: World Wide Web Consortium. http://www.w3.org/TR/xml-infoset/

[Thompson et al. 2001] Thompson, Henry S., David Beech, Murray Maloney, and Noah Mendelsohn. 2001. *XML Schema Part 1: Structures*. W3C Recommendation 2 May 2001. [Cambridge, Sophia-Antipolis, and Tokyo]: World Wide Web Consortium. http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

Notes

[1] Note that the `canonical forms' here described have nothing to do with the W3C canonicalization specification [Boyer 2001].

[2] Note that I am using the term *element* to denote an abstract structural unit, not necessarily a string of characters matching the *element* production in the XML 1.0 grammar. What I am here calling elements, some people prefer to call *element information items*. This is correct and my only objection to it is that it costs the reader and me six extra syllables each time I use it. Wherever there is any danger of confusion, between the abstract element information item and the serialized XML form of the element, I will use phrases like *element information item* or *abstract element information item* and *serialized form*.

[3] It's not clear to me whether the PSVI attributes described here should be in the XSI namespace or a separate one. For now, I simply note that it's a design choice we need to make eventually.

[4] The value is

- "`full`" if this element was strictly assessed and none of its attributes and children has [validation attempted] with any value except "`full`"
- "`none`" if this element was *not* strictly assessed and and none of its attributes and children has [validation attempted] with any value except "`none`"
- otherwise "`partial`"

[5] Informally, the value is

- "`valid`" if this element was strictly assessed, the element is locally valid, and none of its attributes

and children is invalid, and declarations were found where needed for all of its attributes and children

- "`invalid`" if this element was strictly assessed but either it was not locally valid, or some of its attributes or children were invalid, or some declaration was not found, although known to be required
- "`notKnown`" if this element was not strictly assessed.

[6] Or perhaps for all the attributes including those being added in the PSVI serialization -- a design choice to be made.

[7] A coherent proposal could also be built around the idea of separate XML documents for each namespace, but I think the single-document idea is simpler to think about, implement, and use.

[8] This would allow a more satisfactory resolution of the underlying concern of Last-Call issue 121, which proposed that all schema components (elements, types, notations, global attributes, ...) be placed in the same symbol space (and the *name* attributes on their declarations be declared as having type *ID*), in order to ensure that all schema components be easily referenceable by URI. The WG agreed that the goal was important, but was not sold on the proposed solution. We suggested instead the use of XPaths to locate schema components, uncomfortably aware that this requires the user to have more knowledge about the organization of the schema document than we would like to require.

[9] The reverse would also be possible, if we preferred: to retain the use of local declarations.

Either way, it would appear likely that unprefixed local elements may need special treatment; if we find a good way to distinguish between unprefixed local elements and other children, it may also work for local attributes, which are always unprefixed. In that case we may want the dump format not to have any local attributes, either.

[10] This is, I emphasize, solely for illustration; I have not thought enough about it to know how I think references should work.

[11] Some may feel that similarity without identity is more confusing that dissimilarity. The dump format is not wholly the same as the transfer syntax; it might be best, perhaps, to make it wholly different, then.